Research Report 257

Smart Integration: A Test Case Study

**M J Zemerly, E Papaefstathiou, T J Atherton,
D J Kerbyson, G R Nudd**

RR257

This study is concerned with characterisation of a parallel algorithm, namely partial 1D FFT on a parallel system (2 x T800). Analytical expressions for the "execution time" for a single processor and 2 processors are discussed and used to obtain a performance measure (execution time) for the application at hand.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

December 1993

Smart Integration: A Test Case Study*

M. J. Zemerly, E. Papaefstathiou, T. J. Atherton,
D. J. Kerbyson and G. R. Nudd

Department of Computer Science
University of Warwick
Coventry CV4 7AL
email: jamal@dcs.warwick.ac.uk

November 30, 1993

Abstract

This study is concerned with characterisation of a parallel algorithm, namely partial 1D FFT on a parallel system (2 × T800). Analytical expressions for the "execution time" for a single processor and 2 processors are discussed and used to obtain a performance measure (execution time) for the application at hand.

1. Introduction

The smart integration exercise is an additional task to test the PEPS ideology on a small problem and to gain a better understanding of the interactions between the different methods involved in the project. Namely, characterisation, modelling, simulation and possibly benchmarking. This report is concerned with the characterisation issues and will describe in detail the different stages of the characterisation process. Initially, descriptions of the algorithm and the platform will be provided, followed by a description of the characterisation method.

*This work is carried out under the aegis of "Performance Evaluation of Parallel Systems (PEPS)" ESPRIT project no. 6942.

2. Description of the Algorithm

The algorithm selected for this task is a 1D FFT. The reasons for selection of the FFT are its wide usage, possibility of many forms of parallelism and simplicity of implementation and analysis. The method used to calculate the FFT is based on that described in Rabiner and Gold [RG75] and has the following options:

- radix 2 implementation
- decimation in frequency (DIF)
- 2048 32-bit complex input samples
- fixed coefficients $(\sqrt{2}/2, i\sqrt{2}/2)$ to compute the same add-multiply in all butterflies.

The FFT flow chart of the DIF butterfly is given in Figure 1.

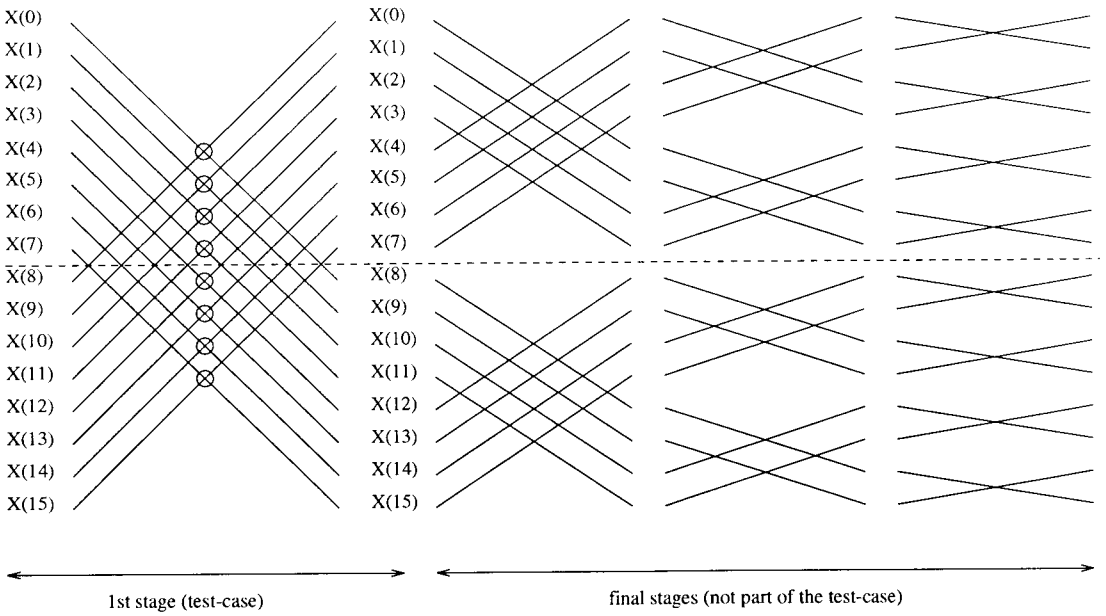

Figure 1. Flow chart of the FFT used (only 16 samples are shown here).

Only the first stage of the FFT is considered here; this is the only stage that requires data communication between processors. The other stages run in parallel without any interaction and do not present any problematic form for performance evaluation study.

Figure 2 shows a representation of a DIF butterfly.

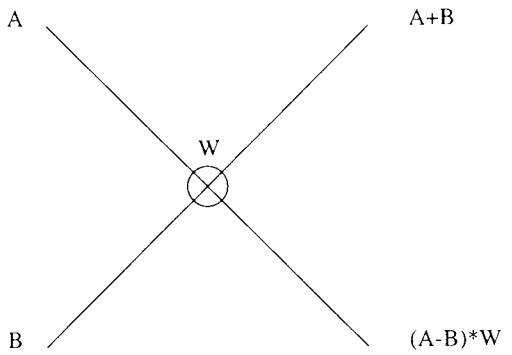

Figure 2. Butterfly for DIF FFT.

The C program listing of the partial DIF FFT used in this exercise is given in Appendix A. Figure 3 shows the computation and dataflow between the 2 processors.

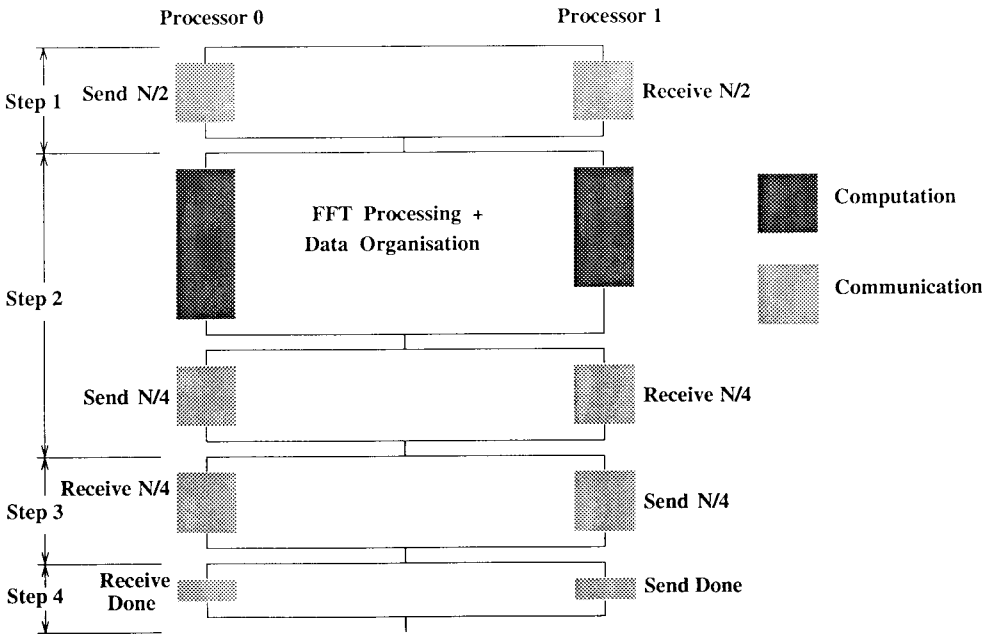

Figure 3. Computation and data flow diagram.

Note that a sequential program is needed to check the results of the characterisation stage. The coefficients can be implemented in 2 possible ways since they are

constants. The first is to have only 2 variables with the values of the coefficients. The second implementation is to compute the sine table (of size 3N/4) required for the number of data samples of the same value and store them in an array and fetch the correct coefficient every time one is needed. This is normally done when computing an FFT. The listing of the FFT function for the second method is given in Appendix B. The 2 implementations will be discussed here.

3. Description of the Hardware

The platform selected is a 2 (4MByte×25MHz) T800 Parsytec machine connected as shown in Figure 4. The data are kept in external memory and the program in internal memory. The transputers have 4 Kbyte on-chip memory. Figure 4 shows a schematic representation of the hardware used.

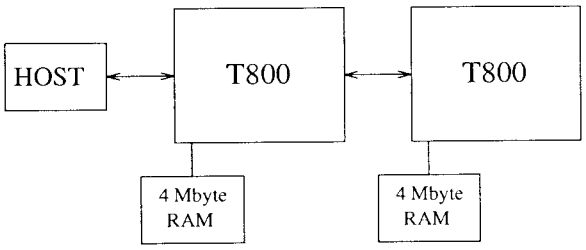

Figure 4. Smart integration hardware.

4. Description of the Characterisation Method

The method used here to characterise the FFT algorithm is based on the method used in Zemerly [Zem93] and Nudd *et al.* [NPP+93]. The "execution time" is selected as a performance measure. For a program running on a single processor the execution time is equal to:

$$\mathrm{T_{TCPU}} = (n_{\mathrm{execution}} + n_{\mathrm{memory}}) \times t \tag{1}$$

where $n_{\mathrm{execution}}$ is the number of clock cycles needed for program execution without the memory effect and n_{memory}, the number of extra clock cycles required for accessing the external memory and can be defined in terms of the number of memory accesses per program, $\mathrm{M_{acc}^{prog}}$, miss penalty and miss rate:

$$n_{\mathrm{memory}} = \mathrm{M_{acc}^{prog}} \times \mathrm{M_{rate}} \times \mathrm{T_{penalty}} \tag{2}$$

Substituting equation (2) in equation (1) and factoring the instruction count i gives:

$$\mathrm{T_{TCPU}} = i \times \left(c_{\mathrm{execution}} + \mathrm{M_{acc}^{inst}} \times \mathrm{M_{rate}} \times \mathrm{T_{penalty}} \right) \times t \tag{3}$$

where $c_{\mathrm{execution}}$ is the cpu clock cycle per instruction excluding the memory effect.

For a program running on a multi-processor there is a penalty of communication and synchronisation between processors. Parallel algorithms can be modelled in two ways according to the computation-communication relationship as described in Basu *et al.* [BSKP90] and Zemerly [Zem93]: overlapped and non-overlapped models. In both models the execution of an algorithm is modelled as repetitive steps where a step consists of a computation followed by a communication. In the overlapped model, computation and communication can be partially or completely overlapped. In the non-overlapped model all the processors do some computation and then synchronise and exchange data.

It is assumed here that all inter-process communication times can be estimated *a priori* and that there are no queuing delays in the system. The data domain is assumed partitioned into sub-domains of equal size and all the processors execute the same program on a different data domain (SPMD).

The smart integration problem can be modelled in either ways but the easier non-overlapped model was selected in the implementation. Figure 5 shows the characterisation model selected for the smart integration exercise.

As can be seen from the smart integration model shown in figure 5 there is one processing and 4 communication modules. 4 steps can be assumed here, only one of them with the processing module and the others without as shown in the figure.

Assuming a perfect load balancing, the execution time T_k for a program running on a k processor system is given by:

$$\mathrm{T}_k = \frac{\mathrm{T_p}}{k} + \mathrm{T_s} + \mathrm{T_{po}} + \sum t_i \tag{4}$$

where $\mathrm{T_p}$ is the execution time (for the part that can be parallelised) of the algorithm on one processor, $\mathrm{T_s}$ is the execution time of the serial part of the algorithm and initialisation time, $\mathrm{T_{po}}$ is the parallel overhead processing required when parallelising an algorithm and t_i is the communication time for "step" i.

The value of t_i can be estimated from the number of bytes transferred in each cycle using:

$$t_i = t_{\mathrm{start-up}} + t_{\mathrm{send}} \times \mathrm{B} \tag{5}$$

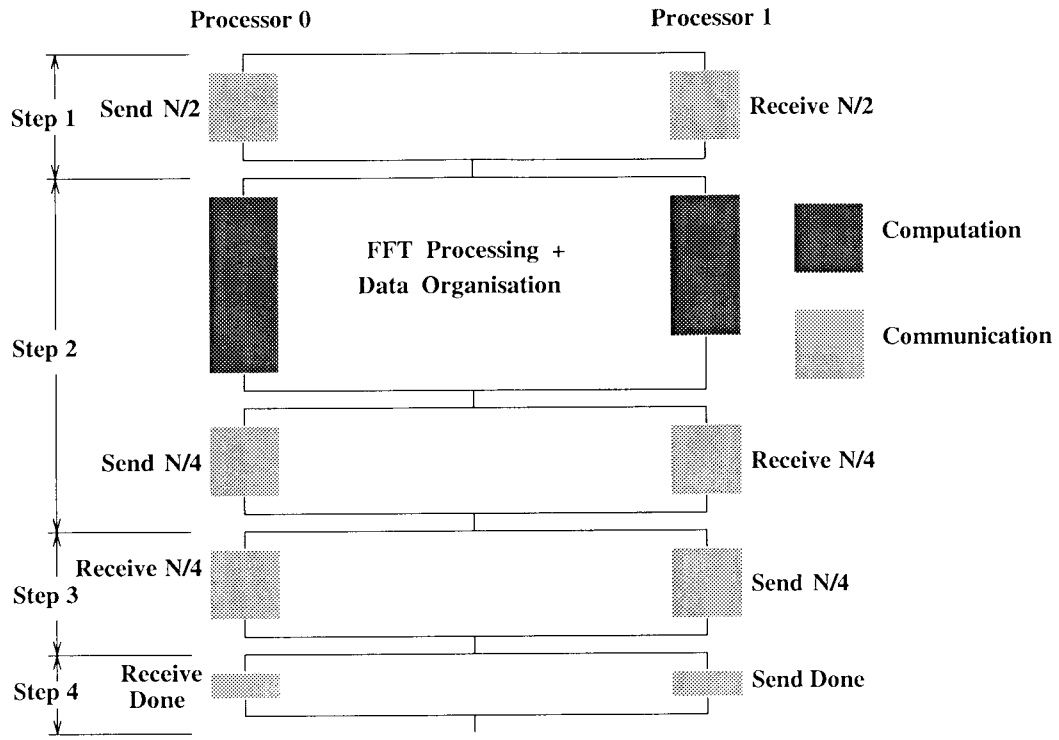

Figure 5. Non-overlap computation-communication model for smart integration.

where $t_{\text{start-up}}$ represents the message start-up overhead or latency, t_{send} the pure communication time, which is inversely proportional to the link bandwidth, and B is the number of bytes transmitted between adjacent processors.

Some of the parameters used in these equations are readily available as the clock cycle time, number of processors and the memory miss rate which is assumed 1 since the data is kept in the external memory. Other parameters have to be measured from the system. These parameters are the number of execution cycles required for the FFT, the number of external memory cycles required to access the data, time penalty for accessing the external memory, either number of instructions in the program and average clock cycle per instruction or number of cycles required to execute the program, communication start-up time, communication transfer rate, initialisation/sequential time of the parallel program (this factor is negligible in the case of smart integration) and the parallel overheads. In the following sections methods to obtain each of these parameters are explained and discussed.

5. Measuring Characterisation Parameters

5.1. Measuring the Program Parameters

First the number of cycles of the fourier transform program is obtained using the following procedure.

1. Create a software execution graph for the function in question. For example for the ffourtr function the execution graph will look something like that:

```
finit (function init)
vinit (Variable init)
   l1 (loop 1)
      l2 (loop 2)
         l3 (loop 3)
            proc (Main processing)
```

The purpose of this graph is to come up with an expression to calculate the total number of cycles. For the above graph the total number of cycles is:

$$C = C(\text{finit}) + C(\text{vinit}) + n1 \times [C(l1) + n2 \times (C(l2) + n3 \times (C(l3) + \text{proc}))] \quad (6)$$

where: n1,n2,n3 are the number of loop execution for l1, l2 and l3 respectively. C() is a function that returns the number of cycles per function.

Here n1 and n2 are 1 where n3 is 512.

2. The next step is to identify the number of cycles per each node of the graph (the C function).

In order to do that the source code for each part is isolated. For example the isolated source code for the loop l3 is:

```
for( cradix=0; cradix < nbradix; cradix ++ )
;
```

Then the source code of the isolated part is compiled with the option -S to produce the assembly listing. If, for example, the file that contains the isolated for loop is called test.c the following compile command was used:

ancc.px -S -c test.c

From the assembly code in the file test.s only the part for the loop execution is isolated excluding function initialisation, return etc. The loop assembly code after the exclusion of irrelevant code looks like:

```
        ldc     0
        stl     .ti1
.L15:
        ldl     .ti1+1
        ldl     .ti1
        gt
        cj      .L12
.L14:
.L13:
        ldl     .ti1
        adc     1
        stl     .ti1
        j       .L15
.L12:
```

The next step is to use the csh script inscnt.csh (see Appendix C) to count how many times the same instruction occurs in the source code. This is done by typing:

csh inscnt.csh

This automatically will filter all the assembly codes in the directory and output the result to the stdout. The result for the above example is shown in Table 1.

times	operation	cycles	total cycles
1	adc	1	1
1	cj	2	2
1	gt	2	2
1	j	3	3
1	ldc	1	1
3	ldl	2	6
2	stl	1	2
total			17

Table 1. Counting the number of cycles

Then using the Transputer data book the number of cycles for each of the command is then extracted and the total number of cycles required to execute the node is then calculated which in this case is 17 cycles. This number includes the the CPU cycles required to access the internal memory. The external memory effect will be discussed later. Since the other loops have exactly the same pattern (int loop counter, constant initialization integer comparison. integer increment) we can conclude that the cycles for all the loops are the same :

C(l1) = C(l2) = C(l3) = 17

The next step is to find the number of cycles for all the primitive operations.

finit : 70 cycles and vinit : 53 cycles

The main butterfly processing loop "proc" is the most important part of processing and the assembly count of operations as well as the number of cycles required for execution are listed in Table 2.

times	operation	Name	cycles	total
7	adc	add constant	1	7
10	add	add	1	10
10	bcnt	byte count	2+1	30
4	dup	duplicate top of stack	1+1	8
1	fpadd	fp add single	6+1+7	14
2	fpldnladdsn	fp load non local and add single	8+1+7	32
4	fpldnlmulsn	fp load non local and mult single	13+1+7	82
10	fpldnlsn	fp load non local single	2+1+7	100
6	fpstnlsn	fp store non local single	2+1+7	60
3	fpsub	fp subtract single	6+1+7	42
2	fpumulby2	fp multiply by 2.0	6+1+1+7	30
26	ldl	load local	2	52
10	ldlp	load local pointer	1	10
6	ldnl	load non-local	2	12
8	stl	store local	1	8
				—
total				497

Table 2. Cycle count for the FFT routine *ffourtr* without sine[3N/4].

In the cycles column in Table 2 extra cycles were added as follows:

- An extra cycle was added if the operation code was more than 4 bit.

- For operands greater than 4 bit an extra cycle was added for every extra 4 bit. So in the case of a floating point operation, 7 cycles were added.

Table 2 shows that the number of cycles required to execute the node proc (one butterfly) is 497 cycles. 17 cycles are also required for the loop iteration.

For example, using equation 6 from step 2 the total number of cycles required to execute the subroutine "ffourtr" for 1024 complex data samples (512 butterflies) without taking into consideration the effect of the external memory is:

$$C = 70 + 53 + 1 \times [17 + 1 \times (17 + 512 \times (17 + 497))] = 263325 \quad \text{cycles}$$

A small problem in finit is that during function initialisation a compiler internal function CRT_stack_extender is called. Since the source code for this function is not available it was assumed that the function is a small housekeeping function and 10 cycles would be sufficient to execute it. In this example this is not a problem since the main loop is the dominant performance factor but in the case of small functions it should be considered as fairly important.

Another loop is required for the second method for computing the sine array. Since the sine source code is not available another program was written to measure the time for the sine execution. A thousand measurements were taken and the average time was 195 μs with average difference of 0.5 μs and standard deviation of 0.25. The number od sines required for N data samples is equal to 3N/4. The number of cycles of the subroutine with the sine array was also obtained in a similar way to method one and is equal to 539 cycles.

For timing the events the transputer instruction ldtimer was used with high priority switched on to give an accuracy of 1 microsecond. This is done because of reported inconsistency with the TimeNowHigh() time function provided by Parix, the Parsytec operating system (see [Gre93]). The timing function is called SysTimer. The same procedure has been followed to identify the cycles for the SysTimer function. This is a very simple case since no execution graph is required because no control statements are included in the code. The number of cycles for SysTimer is 57.

5.2. Effect of the External Memory

The time penalty for the external memory for the T800 transputer is discussed in the Transputer Databook [Inm89]. The Transputer external memory is characterised by extra processor cycles per external memory cycle, e [Inm89]. The

time penalty to access the external memory depends on the value of ϵ which is typically equal to (and in this case assumed to be) 5. The external memory is accessed 22 times per butterfly in the FFT as can be shown from the assembly source code from the number of loading and storing operations of floating point data.

5.3. Measuring the Parallel Overhead Parameters

The function memcpy used to organise the data for processor "0" and processor "1" was measured in a similar fashion to ffourtr and the results are given in Table 3. Table 3 shows the number of cycles required to copy 4 Kbyte.

no.	function	meaning	cycles	total
3	ldc	load constant	1	3
1	ldl	load local	2	2
2	ldlp	load local pointer	1	2
1	move	move message	2w+8+1	2048+8+1
				—
total				2064

Table 3. Timings for the memcpy function for 4 Kbyte.

5 cycles penalty for every read/write should also be added for memcpy. The total is 12304 cycles. Other parameters such as the communication initialisation will be discussed within the communication parameters. Some other miscellaneous instructions such as difference of times and assigning timing variables were estimated at 200 cycles. SysTimer is discussed previously and it takes 57 cycles. SysTimer was called 16 times on processor 0.

5.4. Measuring the Communication Parameters

Table 4 shows the number of cycles required to initialise the links between the 2 processors. Here The call operation has some extra cycles for the function ConnectLink it calls. A program was written to measure the time it takes to connect 2 processors (See Appendix E). This program makes a 100 links and then takes the average of these and that value will be used instead of t(ConnectLink). The program was tested with MakeLink and GetLink as well as ConnectLink and

no.	function	meaning	cycles	total
1	call	function call	7	7
1	cj	conditional jump	2	2
1	dup	duplicate top of the stack	1+1	2
1	eqc	equals constant	2	2
4	ldc	load constant	1	4
2	ldl	load local	2	4
2	ldlp	load local pointer	1	2
4	stl	store local	1	4
1	stnl	store non-local	2	2
				—
total				29

Table 4. Timings for initialisation of links between processors.

and both gave almost the same results (see Table 5).

MakeLink	Proc 0 Time	1112.92 μs	Proc 0 terminates
GetLink	Proc 1 Time	1110.01 μs	Proc 1 terminates
ConnectLink	Proc 0 Time	1132.00 μs	Proc 0 terminates
ConnectLink	Proc 1 Time	1129.15 μs	Proc 1 terminates

Table 5. Timings for MakeLink, GetLink and ConnectLink functions

Since it is ConnectLink which is used in the program the value of 1132 μs will be used.

Also another program was written to measure the communication bandwidth between two processors and the communication start-up time. The program listing is given in Appendix D. In this program, the two processors are first initialised to make a link between themselves. Then a block of data is sent from the first processor to the second and sent back and the time is recorded for send and receive the block on both processors. The bandwidth is calculated using Bandwidth = BlockSize/(1024 $\times$ 1024 $\times$ 0.5 $\times$ time). BlockSize was increased in powers of 2 from 4 to 262144 byte and a bandwidth value was obtained for every block size. A plot of the bandwidth vs block size is shown in Figure 6.

Note that the bandwidth is variable because it includes the start-up time and transfer time. The communication start-up time was obtained by fitting a straight

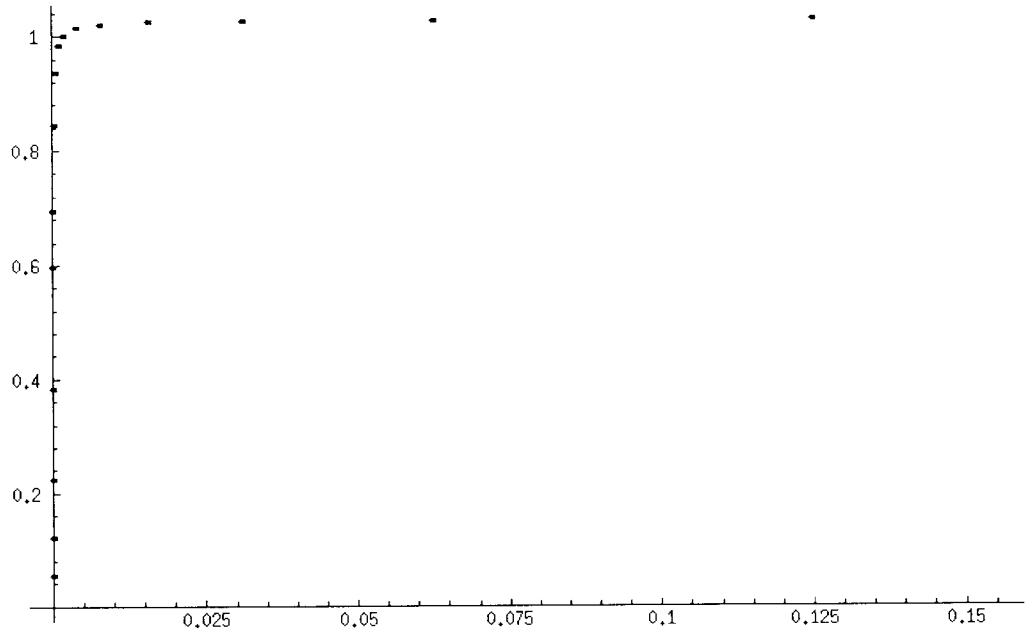

Figure 6. Relationship between block size and bandwidth.

line to half the time vs block size data and finding the time at block size of 0. Figure 7 shows the relationship between block size and time.

The plot only shows up to block size 32 byte to show the start-up time at zero block size clearly. The equation obtained from the fit was (units in seconds and Mbyte):

Communication time = start-up time + transfer time

= 0.0000507866 + 0.973054 blocksize

Note that transfer time is equal to the block size divided by the transfer rate (B/R). The transfer rate is then equal to (1/0.973054 = 1.03 Mbyte/s) The startup time is rounded-off to 51 μs.

6. Program Time Measurement

The program time measurement were taken between different stages of the program and the times recorded for these stages are given on the standard output and saved in a file. These times measured in μs are shown in Table 6.

As can be seen from Table 6 the butterfly processing took about 13251 μs on processor 0 and 13296 on processor 1. The total execution times were 31269 and

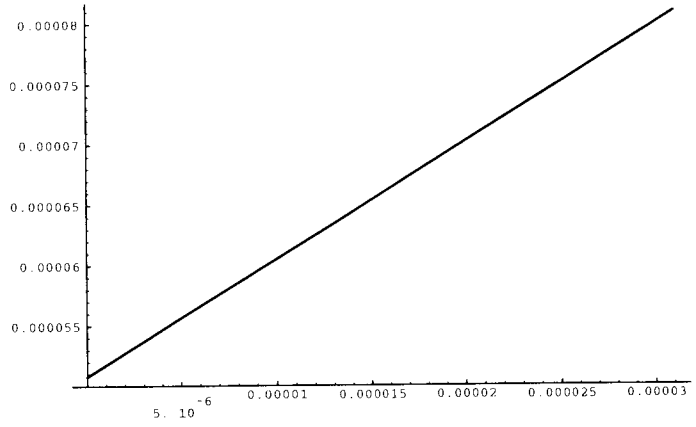

Figure 7. Relationship between block size and time.

32573 μs for processors 0 and 1 respectively. The program was also run on one processor and the time recorded for executing the program was 26515 μs. Note that the time taken for the butterfly processing on one processor (26515 μs) is less than the time taken on 2 processors. That is because the communication overhead exceeded the processing time.

For the sine computing method the sequential time was 335592 μs and the time measurements on 2×T800 are given in Table 7.

It is also worth noting that in both cases the total times were fluctuating in the order of ± 30 μs but this was ignored and only one reading was selected.

7. Predicted Times Without Computing Sine[3N/4]

7.1. Predicted Sequential Time

The predicted time for the FFT running on one processor is given by:

$$T = (n_{execution} + n_{memory}) \times t$$

$$n_{memory} = [T_h + M_{ratio} \times T_{penaly}] \times M_{acc}^{prog}$$

where T_h is the average hit time (in this case 1), M_{ratio} is the internal RAM miss ratio (assumed 1), T_{penaly} is the time penalty to access the external memory and M_{acc}^{prog} is the number of external memory accesses per program.

```
Processor 0 received DONE signal (1) from slave
Processor 1 DONE=1 time_start=1882037 time_end=1914610
Processor 0 DONE=1 time_start=4963750 time_end=4995019
Processor 0 finished its task in 31269 us
Processor 1 finished its task in 32573 us
Processor 0 finished initialisation of links in 1124 us
Processor 1 finished initialisation of links in 2144 us
Processor 0 finished sending half of the data in 7536 us
Processor 0 finished arranging its data in 759 us
Processor 1 finished receiving half of the data in 8618 us
Processor 0 finished butterfly processing in 13251 us
Processor 1 finished butterfly processing in 13296 us
Processor 0 finished sending a quarter of the data in 3779 us
Processor 0 finished receiving a quarter of the back in 3936 us
Processor 1 finished arranging data for transfer in 382 us
Processor 0 finished receiving FINISHED signal in 67 us
Processor 1 finished receiving a quarter of the data in 4114 us
Processor 1 finished sending FINISHED signal in 53 us
Processor 0 times for arranging data for other processor (760)
            and itself (759)
```

Table 6. Timings for the FFT program (without computing the sine) on 2 T800s.

$$n_{execution} = A + butterfly\_cycles \times Butterflies$$

where Butterflies is the number of the butterflies and A is the extra cycles used for timing the events, routine initialisation and variable intialisation and in this case is equal to $57 + 70 + 53 + 17 + 17 = 214$ cycles.

$$n_{memory} = (1 + 5) \times 16 \times Butterflies$$

This gives:

$$T = (214 + 514 \times 1024 + 6 \times 22 \times 1024) \times 40 \times 10^{-9} = 26468\mu s$$

Compared to the measured time for butterfly processing, 26468 μs, the predicted time, 26635 μs, is 167 μs (-0.63%) slower.

```
Processor 1 DONE=1 time_start=878436 time_end=1060071
Processor 0 DONE=1 time_start=5299160 time_end=5479490
Processor 0 finished its task in 180330 us
Processor 1 finished its task in 181635 us
Processor 0 finished initialisation of links in 1125 us
Processor 1 finished initialisation of links in 2146 us
Processor 0 finished sending half of the data in 7819 us
Processor 1 finished receiving half of the data in 8902 us
Processor 0 finished arranging its data in 759 us
Processor 0 finished butterfly and sine processing in 161929 us
Processor 1 finished butterfly and sine processing in 161925 us
Processor 0 finished sending a quarter of the data in 3930 us
Processor 0 finished receiving a quarter of the data in 3883 us
Processor 1 finished arranging data for transfer in 383 us
Processor 0 finished receiving FINISHED signal in 67 us
Processor 0 times for arranging data for other processor (761)
            and itself (759)
Processor 1 finished receiving a quarter of the data in 4314 us
Processor 1 finished sending a quarter of the data in 3870 us
Processor 1 finished sending FINISHED signal in 52 us
```

Table 7. Timings for the FFT (with sine[3N/4]) program on 2 T800s.

7.2. Predicted Parallel Time

Recalling equation 4, the predicted time for the FFT running on two processors (values are taken for processor 0) is given by:

$$T_k = \frac{T_p}{k} + T_s + T_{po} + \sum_{i=1}^{4} t_i$$

For simplicity and because it is too small compared to the total time, the effect of T_s is neglected here and it is assumed that T_p is equal to T (the sequential time on one processor).

$$T_{po} = C(\text{SysTimer}) \times \text{Timings} + C(\text{misc}) + C(\text{InitLink}) + 4 \times C(\text{memcpy(4Kb)}$$

$$T_{po} = 57 \times 15 + 200 + (29 + C(\text{ConnectLink})) + 4 \times 12304 \quad \text{cycles}$$

$$= \frac{50300}{25} + \mathrm{t(ConnectLink)} = 2012 + 1132 = 3144\mu s$$

$$\sum_{i=1}^{4} \mathrm{t_i} = \mathrm{t(8Kbyte)} + \mathrm{t(4Kbyte)} + \mathrm{t(4Kbyte)} + \mathrm{t(4byte)} + 4 \times \mathrm{t_{start-up}}$$

$$\sum_{i=1}^{4} \mathrm{t_i} = \frac{8}{1024 \times \mathrm{R}} + \frac{4}{1024 \times \mathrm{R}} + \frac{4}{1024 \times \mathrm{R}} + \frac{4}{1024 \times 1024 \times \mathrm{R}} + 4 \times 51$$

$$= 7585 + 3793 + 3793 + 4 + 4 \times 51 = 15379\mu s$$

Substituting in equation 4 gives:

$$\mathrm{T_k} = \frac{26468}{2} + 3144 + 15379 = 31757\mu s$$

The time for processor 0 to finish is 31757 μs. A difference of only 488 μs (1.56%) from the measured time of 31269 μs.

8. Predicted Times with Computing Sine[3N/4]

8.1. Predicted sequential Time

$$\mathrm{n_{execution}} = \mathrm{A} + \mathrm{butterfly\_cycles} \times \mathrm{Butterflies} + \mathrm{C(sine)}$$

butterfly_cycles=539+17=556 cycles

finit=70 cycles and vinit=153 cycles.

A = 57+70+153+17+17+29 = 343 cycles

$$\mathrm{t(sine)} = (3 \times \mathrm{DataSamples}/4) \times (\mathrm{SineTime} + \mathrm{LoopTime})$$
$$= (3 \times 2048/4) \times (195 + 17/25) = 300564\mu s$$

$$\mathrm{C(Butterflies)} = 1024 \times (556 + 6 \times 22) = 704512\mathrm{cycles}$$

$$\mathrm{T} = (343 + 704512) \times 40 \times 10^{-9} + 300564 = 28194 + 300564 = 328758\mu s$$

8.2. Predicted Parallel Time

$$C(\text{Butterflies}) = 512 \times (556 + 6 \times 22) = 512 \times 688 = 352256\text{cycles}$$

$$C(\text{sine}) = (3 \times \text{DataSamples}/4) \times (\text{SineTime} + \text{LoopTime})$$

$$= (3 \times 1024/4) \times (195 + 17/25) = 150282\mu s$$

$$T_{po} = 3144\mu s$$

$$\sum_{i=1}^{4} t_i = 15379\mu s$$

$$T_k = \frac{328758}{2} + 3144 + 15379 = 182902\mu s$$

9. Conclusion

The predicted times obtained from the characterisation model used gave similar answers to the measured times. The differences for the sequential and parallel 1D FFT programs (with and without computing the sine array) between the predicted and the measured time were less than 4%. Table 9. summarises the results obtained.

	Without sine[3N/4]			With sine[3N/4]		
	Sequential	Parallel	Sk	Sequential	Parallel	Sk
Measured	26635	31269	0.85	335592	180330	1.86
Predicted	26468	31757	0.83	328758	182902	1.80
Difference	-167	488	0.02	-6834	2572	-0.06
% Difference	-0.67	1.56	2.35	-2.04	1.43	3.23

A Listing of the DIF FFT C program

```
------------------------------------------------------------
/*

Project    : PEPS
Task    : Smart Integration
Program    : fft.c
Purpose    : FFT of 2048 complex points (radix 2) on 2 T800s
Authors    : Jamal Zemerly and Efstathios Papaefstathiou
Date    : 1/9/93

Description
-----------

The method used to compute the fft is the bit-reversed on the
outputs (DIF, Rabiner and Gold, 1975, page 574, Fig 10.1).
2 T800s are used in the computation. First the data is
distributed between the processors (each get N/2 points)
followed by FFT computation and Exchange of data between the
processors (each send N/4 points). The process stops when the
data is received and written in the memory of each processor.

*/

#include<stdio.h>
#include<math.h>
#include<errno.h>

#include<sys/root.h>
#include<sys/sys_rpc.h>

/* Library references */
extern char *optarg;
extern double atof();
extern int atoi();
extern int getopt();
extern char* memcpy();

#ifndef FALSE
#define FALSE 0
```

```
#define TRUE 1
#endif

#define MASTER 0
#define FINISHED 1

#define GETID() (GET_ROOT()->ProcRoot->MyProcID)
#define GETNP() (GET_ROOT()->ProcRoot->nProcs)
#define DNODE 4096
#define NODE 2048
#define HNODE 1024
#define QNODE 512
#define LNODE 8192
#define LHNODE 4096

/* Local function prototypes */
void    Mainmaster(int,int,FILE*);
void    Mainslave(int);
void ffourtr(float*,int);
void    usage(char*);
unsigned int SysTimer(void);
int ChangePriority(int);

main(int argc, char** argv)
{
int nn=NODE;/* number of input data points*/
int p=2; /*number of processors*/
char c;
FILE *fd; /*file descriptor*/
/*
 * read_args - Read arguments
 */

        while ((c = getopt(argc,argv,"Uk:n:f:")) != -1)
        switch (c) {
        case 'k' : p = atoi(optarg);
                   break;
        case 'n' : nn = atoi(optarg);
                   break;
case 'f' : if ((fd = fopen(optarg, "r"))==0)
      printe("bad filename or argument");
```

```
   break;
        case 'U' : usage(" ");
                   break;
}

if(p>2){
   printe("number of processors must be 1 or 2");
   AbortServer(1);
}

if(p==2){
   if (GETID() == MASTER)
Mainmaster(nn,p,fd);
   else
Mainslave(nn);
/*    AbortServer(0);*/
}
if(p==1){
   if (GETID() == MASTER){
Mainmaster(nn,p,fd);
   }
   AbortServer(0);
}
}

/* Master transputer operation*/

void Mainmaster(int nn, int p, FILE *fd)
{
int i,j;
int hn=nn/2;
int qn=nn/4;
int dn=2*nn;
int ln=nn*4;
int lhn=hn*4;
int lqn=qn*4;
int ldn=dn*4;
float buf[NODE], data[DNODE];
static LinkCB_t* ComLink; /* Client virtual link descriptor*/
int err, pr;
unsigned int time1, time2, time3, time4, time5, ctimeextra[5];
```

```
unsigned int time6, time7, time8, time9, time10, time11;
unsigned int timeall, ctime1, ctime2, ctime3, ctime4;
unsigned int ctime5, ctime6, comptime;
char *DONE;
j=0;

/*change priority of the algorithm to high*/

pr=ChangePriority(HIGH_PRIORITY);

/*copy the opened file into data*/

memcpy(&data, fd, sizeof(float)*DNODE);

time1=SysTimer();

/* Initialise the links between the 2 processors*/
if(p==2){
   if((ComLink=ConnectLink(1,1000,&err))== NULL){
      printe("Connect link error %d Proc %d \n", err, GETID());
      AbortServer(0);
   }
   time2=SysTimer();
   ctime1=time2-time1;

/*
 * Organise half the data for the other processor
 * This could be avoided with sending twice quarter of the data
 */
   memcpy(&buf[0], &data[HNODE], sizeof(float)*HNODE);
   memcpy(&buf[HNODE], &data[HNODE+NODE], sizeof(float)*HNODE);

/*send half the data to the other processor*/

   time3=SysTimer();
   ctimeextra[j++] = (time3-time2);
   time3=SysTimer();

   if(SendLink(ComLink, buf, LNODE) != LNODE){
     printe("Send failed %d Proc %d \n", errno, GETID());
     AbortServer(0);
```

```
    }

    time4=SysTimer();
    ctime2=time4-time3;
    time4=SysTimer();

/*organise the data for the master in buf*/

    memcpy(&buf[0], &data[0], sizeof(float)*HNODE);
    memcpy(&buf[HNODE], &data[NODE], sizeof(float)*HNODE);

    time5=SysTimer();
    ctime3= time5 - time4;
    ctimeextra[j++] = ctime3;
    time5=SysTimer();

/* do the butterfly processing*/

    ffourtr(buf, HNODE);

    time6=SysTimer();
    comptime=time6-time5;
    time7=SysTimer();

/* send a quarter of the data back to other processor*/

    if(SendLink(ComLink, &buf[HNODE], LHNODE) != LHNODE){
      printe("Send failed %d Proc %d \n", errno, GETID());
      AbortServer(0);
    }

    time8=SysTimer();
    ctime4=time8-time7;
    time8=SysTimer();

/* receive a quarter of the data from the other processor*/

    if(RecvLink(ComLink, &buf[HNODE], LHNODE) != LHNODE){
      printe("Receive failed %d Proc %d \n", errno, GETID());
      AbortServer(0);
    }
```

```
   time9=SysTimer();
   ctime5=time9-time8;
   time9=SysTimer();

/* receive finished signal from the other processor*/

   if(RecvLink(ComLink, DONE, 1) != 1){
      printe("Receive finished failed %d Proc %d \n", \
              errno, GETID());
      AbortServer(0);
   }

   time10=SysTimer();
   ctime6=time10-time9;

   if(*DONE==FINISHED){
     time11=SysTimer();
     printf("Processor %d received DONE signal (%d) \
            from slave \n", GETID(), *DONE);
   }

   timeall=time11-time1;

/*print timing results*/

printf("Processor %d DONE=%d time_start=%d time_end=%d \n", \
        GETID(), *DONE, time1, time11);
printf("Processor %d finished its task in %d us \n", \
        GETID(), timeall);
printf("Processor %d finished initialisation of links in \
        %d us \n", GETID(), ctime1);
printf("Processor %d finished sending half of the data in \
        %d us \n", GETID(), ctime2);
printf("Processor %d finished arranging its data in \
        %d us \n", GETID(), ctime3);
printf("Processor %d finished butterfly processing in \
        %d us \n", GETID(), comptime);
printf("Processor %d finished sending a quarter of the data in \
        %d  us \n", GETID(), ctime4);
printf("Processor %d finished receiving quarter of the data in \
```

```
        %d us \n", GETID(), ctime5);
printf("Processor %d finished receiving FINISHED signal in \
        %d us \n", GETID(), ctime6);
printf("Processor %d arranged data for processor 1 (%d) and for \
        itself (%d) \n", GETID(), ctimeextra[0], ctimeextra[1]);
}

if(p==1){ /*if only one processor is requested*/
   time1=SysTimer();
   ffourtr(data,nn);
   time2=SysTimer();
   timeall=time2-time1;
   printf("Processor %d finished its task in %d us \
                \n", GETID(), timeall);
}


/*change priority to low*/

pr=ChangePriority(LOW_PRIORITY);
}
/* Slave transputer operation*/
void Mainslave(int nn)
{
int i;
int hn=nn/2;
int qn=nn/4;
int dn=2*nn;
int ln=nn*4;
int lhn=hn*4;
int lqn=qn*4;
int ldn=dn*4;
float buf[NODE], buf1[HNODE];
static LinkCB_t* ComLink; /* Client virtual link descriptor*/
int err, pr; /*error and priority change variables*/
unsigned int time1, time2, time3, time4, time5;
unsigned int time6, time7, time8, time9, time10, time11;
unsigned int timeall, ctime1, ctime2, ctime3, ctime4;
unsigned int ctime5, ctime6, comptime;
char *DONE;
```

```
/*change priority to high*/

pr=ChangePriority(HIGH_PRIORITY);

/*start timing from here*/

time1=SysTimer();

/* Initialise the links between the 2 processors*/

 if ((ComLink=ConnectLink(0,1000,&err))== NULL){
        printe("Comm. init. error %d Proc %d \n", err, GETID());
        AbortServer(0);
 }

time2=SysTimer();
ctime1=time2-time1;
time2=SysTimer();

/* receive half the data from the Master*/

if(RecvLink(ComLink, buf, LNODE) != LNODE){
        printe("Receive failed %d Proc %d \n", errno, GETID());
        AbortServer(0);
}

time3=SysTimer();
ctime2=time3-time2;
time3=SysTimer();

/* do the butterfly operations on the data*/

ffourtr(buf, HNODE);

time4=SysTimer();
comptime = time4-time3;
time4=SysTimer();

/* prepare the data for transfer*/

memcpy(&buf1[0], &buf[0], sizeof(float)*HNODE);
```

```
time5=SysTimer();
ctime3= time5-time4;
time5=SysTimer();

/* receive a quarter of the data from the Master*/
        if(RecvLink(ComLink, &buf[0], LHNODE) != LHNODE){
           printe("Receive failed %d Proc %d \n", errno, GETID());
           AbortServer(0);
}

time6=SysTimer();
ctime4 = time6-time5;
time6=SysTimer();

/* send a quarter of the data back to the Master*/

     if(SendLink(ComLink, buf1, LHNODE) != LHNODE){
printe("Send failed %d Proc %d \n", errno, GETID());
AbortServer(0);
}
time7=SysTimer();
ctime5= time7-time6;

/* send DONE signal back to the Master*/

*DONE=FINISHED;

time7=SysTimer();

     if(SendLink(ComLink, DONE, 1) != 1){
        printe("Send DONE failed %d Proc %d \n", errno, GETID());
        AbortServer(0);
     }

time8=SysTimer();
ctime6=time8-time7;

timeall=time8-time1;

printf("Processor %d DONE=%d time_start=%d time_end=%d \n", \
```

```
        GETID(), *DONE, time1, time9);
printf("Processor %d finished its task in \
        %d us \n", GETID(), timeall);
printf("Processor %d finished initialisation of links in \
        %d us \n",  GETID(), ctime1);
printf("Processor %d finished receiving half of the data in \
        %d us \n", GETID(), ctime2);
printf("Processor %d finished butterfly processing in \
        %d us \n", GETID(), comptime);
printf("Processor %d finished arranging data for transfer in \
        %d us \n", GETID(), ctime3);
printf("Processor %d finished receiving quarter of the data in \
        %d us \n", GETID(), ctime4);
printf("Processor %d finished sending a quarter of the data in \
        %d us \n", GETID(), ctime5);
printf("Processor %d finished sending FINISHED signal in \
        %d us \n", GETID(), ctime6);

/*change priority back to low*/
pr=ChangePriority(LOW_PRIORITY);
}

/*Butterfly processing routine*/

void ffourtr(data,nn)
float data[NODE]; /*data string*/
int nn;  /* number of data items*/
{
int radix=2;
int i, m, n, cstage, cblock, cradix, nbstage, nbblock;
int nbradix, mblock;
float tr, ti; /*real and imaginary parts of T*/
float wr=0.707;
float wi=0.707;
unsigned int time1, time2;

nbblock=1; /* start with 1 and increase by ^2 for each block*/
nbradix=nn/2; /* reduce by factor of 2 for each stage*/
nbstage=1; /* log2(nn)-1 but for smart integration it is 1*/

m=0;
```

```
n=m+nn;
for (cstage=0; cstage<nbstage; cstage++){

        for (cblock=0; cblock<nbblock; cblock++){

for (cradix=0; cradix<nbradix; cradix++){
data[m] = data[m] + data[n];
tr = data[m] - 2*data[n];
data[m+1] = data[m+1] + data[n+1];
ti = data[m+1] - 2*data[n+1];
data[n] = tr*wr + ti*wi;
data[n+1] = tr*wi - ti*wr;
m += 2;
n += 2;
                }
        }
}
}
/*
 * usage - Print usage message
 */
void usage(char* mess)
{
        printe("Usage : %s\n",mess);
        printe("\t-k number of processors\n");
        printe("\t-n number of points\n");
        AbortServer(0);

}
```

B Listing of the FFT routine with the Sine Computation

```
-----------------------------------------------------------------

/*partial fft subroutine on 2 processors*/

void ffourtr(data,nn)
float data[NODE]; /*data string*/
int nn;  /* number of data items*/
{

int radix=2;
int i, m, n, cstage, cblock, cradix, nbstage, nbblock;
int nbradix, mblock;
float tr, ti; /*real and imaginary parts of T*/
int ksin, kcos;
int offset=nn/4; /* offset between sin and cos*/
int sinelength = 3*HNODE/4;
float SIN[3*HNODE/4];

nbblock=1; /* start with 1 and increase by ^2 for each block*/
nbradix=nn/2; /* reduce by power of 2 for each radix*/
nbstage=1; /* (log(2) nn)-1 but here it is 1*/

for (i=0; i<sinelength; i++)
        SIN[i]=sin(3*PI/4);

m=0;
n=m+nn;
mblock=m;
ksin=0;
kcos=ksin+offset;

for (cstage=0; cstage<nbstage; cstage++){

        for (cblock=0; cblock<nbblock; cblock++){

                for (cradix=0; cradix<nbradix; cradix++){
                        data[m] += data[n];
                        tr = data[m] - 2*data[n];
                        data[m+1] += data[n+1];
```

```
                        ti = data[m+1] - 2*data[n+1];
                        data[n] = tr*SIN[ksin] + ti*SIN[kcos];
                        data[n+1] = tr*SIN[kcos] - ti*SIN[ksin];
                        m += 2;
                        n += 2;
                        ksin += nbblock;
                        kcos += nbblock;
                }
                mblock=n;
                m=mblock;
                n=mblock +nbradix*2;
                ksin=0;
                kcos=ksin+offset;
        }
}
}
```

C Listing of the Shellscript "inscnt.csh"

```
------------------------------------------------------------------
/*

Project     : PEPS
Task     : Smart Integration
Program     : inscnt.csh
Purpose     : Count the number of instructions of assembly files
Date     : 1/9/93
Authors     : Efstathios Papaefstathiou

Description
-----------

#
# inscnt.csh
# Count the number of transputer instructions of one or more
# assembly files (.s) produced by the parix compilers
#

foreach f (*\.s)
 echo $f
 awk '$1 !~ /^[\._]/ && $1 !~ /^[ \t]*$/ { print "\t" $1 }' $f|\
 sort|uniq -c
end

------------------------------------------------------------------
```

D Listing of the Communication Rate Program

```
------------------------------------------------------------------
/*

Project     : PEPS
Task     : Smart Integration
Program     : comms.c
Purpose     : measure the transfer rate between 2 processors
Date     : 1/9/93
Authors     : Jamal Zemerly and Efstathios Papaefstathiou

Description
-----------
In this program, the two processors are first initialised to
make a link between themselves. Then a block of data is sent
from the first processor to the second and sent back and the
time is recorded for sending and receiving the block on both
processors. The bandwidth is calculated using:

                Block-size
Bandwidth= --------------------
           (1024*1024)*0.5*time

Block size is being increased from 4 to 262144 bytes and a
bandwidth value was obtained for all block sizes.
*/

#include<stdio.h>
#include<math.h>
#include<errno.h>

#include<sys/root.h>
#include<sys/sys_rpc.h>
#include<sys/select.h>

/* Library references */
extern char *optarg;
extern double atof();
extern int atoi();
extern int getopt();
```

```
#ifndef FALSE
#define FALSE 0
#define TRUE 1
#endif

#define MASTER 0
#define FINISHED 1

#define GETID() (GET_ROOT()->ProcRoot->MyProcID)
#define GETNP() (GET_ROOT()->ProcRoot->nProcs)
#define DNODE 4096
#define NODE 2048
#define HNODE 1024
#define QNODE 512
#define LNODE 8192
#define LHNODE 4096

/* Local function prototypes */
void    Mainmaster(int,int);
void    Mainslave(int);
void    usage(char*);
unsigned int SysTimer(void);
int ChangePriority(int);
void TimeWaitHigh(unsigned int);

main(int argc, char** argv)
{
int nn=1;/* number of input data points*/
int p=2; /*number of processors*/
char c;
/*
 * read_args - Read arguments
 */

        while ((c = getopt(argc,argv,"U:n:")) != -1)
        switch (c) {
        case 'n' : nn = atoi(optarg);
                   break;
        case 'U' : usage(" ");
                   break;
```

```
}

if (GETID() == MASTER)
Mainmaster(nn,p);
else
Mainslave(nn);
/* AbortServer(0);*/
}

/* Master transputer operation*/

void Mainmaster(int nn, int p)
{
int i;
float *data;
unsigned int *time, *ctime, *timest, *ctimest;

/*communications variables/structures*/

static LinkCB_t* ComLink; /* Client virtual link descriptor*/

int err, pr; /*error and priority change value*/
unsigned int time1, time2, time3, time4, ctime1, ctime2;
int dn, ldn;
float rate;

pr=ChangePriority(HIGH_PRIORITY);

/*calloc for time and time difference*/

if((time=(unsigned int *) calloc(40,4))==0)
printe ("cannot allocate space for time");

if((ctime=(unsigned int *) calloc(20,4))==0)
        printe ("cannot allocate space for ctime");

timest=time;
ctimest=ctime;


*time++ = SysTimer();
```

```
/* Initialise the links between the 2 processors*/

if((ComLink=ConnectLink(1,1000,&err))== NULL){
  printe("Connect link error %d Proc %d \n", err, GETID());
  AbortServer(0);
}
*time++ = SysTimer();

*ctime++ = *(time-1) - *(time-2);

while(nn<=65536){

dn=nn;

ldn=dn*4;

if((data=malloc(ldn))==0)
printe ("cannot allocate space for data");

*time++=SysTimer();
   if(SendLink(ComLink, data, ldn) != ldn){
      printe("Send failed %d Proc %d \n", errno, GETID());
      AbortServer(0);
   }
   if(RecvLink(ComLink, data, ldn) != ldn){
      printe("Receive failed %d Proc %d \n", errno, GETID());
      AbortServer(0);
   }

*time++ = SysTimer();

*ctime++ = *(time-1) - *(time-2);

/*print timing results*/

free(data);
nn= nn*2;

}
```

```
nn=1;
time=timest;
ctime=ctimest;
i=1;
printf("Processor %d initialised the links in %d us \n", \
        GETID(), *(ctime++));

while(nn<=65536){
ldn=nn*4;
rate=((float)ldn*1000000.0)*2/((float)*(ctime++)*1024.0*1024.0);
printf("%d- Processor %d finished sending %d bytes of data in \
%d us (rate=%f Mbytes/s)\n", i, GETID(), ldn, *(ctime-1), rate);
nn=nn*2;
i++;
}

TimeWaitHigh(10000000);

/*change priority to low*/

pr=ChangePriority(LOW_PRIORITY);

}

/* Slave transputer operation*/
void Mainslave(int nn)
{
int i,j;
int dn, ldn;
float *data;
/*communications variables/structures*/

static LinkCB_t* ComLink; /* Client virtual link descriptor*/
Option_t opt1; /*option of start receive*/

int err;
int pr; /*priority change value*/
unsigned int *time, *ctime, *timest, *ctimest;
unsigned int time1, time2, time3, time4;
unsigned int ctime1, ctime2;
float rate;
```

```
/*change priority to high*/
pr=ChangePriority(HIGH_PRIORITY);

/*calloc for time and time difference*/

if((time=(unsigned int *) calloc(40,4))==0)
        printe ("cannot allocate space for time");

if((ctime=(unsigned int *) calloc(20,4))==0)
        printe ("cannot allocate space for ctime");

timest=time;
ctimest=ctime;
*time++ = SysTimer();

/* Initialise the links between the 2 processors*/

 if ((ComLink=ConnectLink(0,1000,&err))== NULL){
      printe("Comm. initialisation error %d Proc %d \n", \
              err, GETID());
      AbortServer(0);
 }

*time++ = SysTimer();
*ctime++ = *(time-1) - *(time-2);

while(nn<=65536){
dn=nn;
ldn=dn*4;

if((data=malloc(ldn))==0)
           printe ("cannot allocate space for data");


/*start timing from here*/

*time++ = SysTimer();

/* receive the data from the Master*/
```

```
if(RecvLink(ComLink, data, ldn) != ldn){
   printe("Receive failed %d Proc %d \n", errno, GETID());
   AbortServer(0);
}

/*Then send it back -ping-pong */

if(SendLink(ComLink, data, ldn) != ldn){
   printe("Receive failed %d Proc %d \n", errno, GETID());
   AbortServer(0);
}

*time++ = SysTimer();
*ctime++ = *(time-1) - *(time-2);

free(data);

nn = nn*2;
}
nn=1;
time=timest;
ctime=ctimest;
i=1;
printf("Processor %d initialised the links in %d us \n", \
        GETID(), *(ctime++));

while(nn<=65536){
ldn=nn*4;
rate=((float)ldn*1000000.0)*2/((float)*(ctime++)*1024.0*1024.0);
printf("%d- Processor %d finished receiving %d bytes of data in \
%d us (rate=%f Mbytes/s)\n", i, GETID(), ldn, *(ctime-1), rate);
nn=nn*2;
i++;
}
pr=ChangePriority(LOW_PRIORITY);
}

/*
 * usage - Print usage message
 */
void usage(char* mess)
```

```
{
        printe("Usage : %s\n",mess);
        printe("\t-n number of points\n");
        AbortServer(0);

}
```

E Listing of the Communication Initialisation Program

```
------------------------------------------------------------
/*
Project   : PEPS
Task      : Smart Integration
Program   : bcon.c
Purpose   : Check connection time using Makelink and Getlink.
            This version works only with adjacent transputers.
            ConnectLink behaves exactly the same.
Author    : Efstathios Papaefstathiou
Date      : 4/10/93

Description
-----------
This program measures the time taken to connect 2 processors.
It  makes a 100 links and then takes the average of these
and that value will be used in the characterisation.
The program was tested with MakeLink and GetLink as well as
ConnectLink and both gave almost exactly the same results.
It is ConnectLink which is used in the FFT program.

*/

#include <stdio.h>

#include <sys/root.h>
#include <sys/sys_rpc.h>

#define MAX_LOOP (100)

extern unsigned SysTimer(void);
void Main0(void),Main1(void);

main()
{
if( GET_ROOT()->ProcRoot->MyProcID == 0 )
Main0();
else
Main1();
```

```
printf("Proc %d terminates\n",
GET_ROOT()->ProcRoot->MyProcID);
}

void Main0(void)
{

unsigned int tcnt,i,j;
int err;
LinkCB_t *link;

/* Check if the grid is 1x2 */
if( GET_ROOT()->ProcRoot->DimX != 1 ||
    GET_ROOT()->ProcRoot->DimY != 2 ) {
printe("Usage: Grid must be 1x2\n");
AbortServer(1);
}

/* Change priority to high */
ChangePriority(HIGH_PRIORITY);

/* Initially connect 0 with 1 */
tcnt = 0;

for( i = 1; i <= MAX_LOOP ; i++ ) {
j = SysTimer();
link = MakeLink(1,i,&err);
tcnt += SysTimer()-j;

if( link == NULL ) {
   printf("MakeLink error(%d) from 0\n",err);
   AbortServer(1);
}
BreakLink(link);
}

printf("MakeLink Proc 0 Time %10.2f\n",
(float)tcnt/(float)MAX_LOOP);
}
```

```
void Main1(void)
{
unsigned int tcnt,i,j;
int err;
LinkCB_t *link;

/* Change priority to high */
ChangePriority(HIGH_PRIORITY);

/* Initially connect 0 with 1 */
tcnt = 0;

for( i = 1; i <= MAX_LOOP ; i++ ) {
j = SysTimer();
link = GetLink(0,i,&err);
tcnt += SysTimer()-j;

if( link == NULL ) {
   printf("GetLink error(%d) from 1\n",err);
   AbortServer(1);
}
BreakLink(link);
}

printf("GetLink Proc 1 Time %10.2f\n",
(float)tcnt/(float)MAX_LOOP);

}
```